Anomaly Detection in Large Sets of High-Dimensional Symbol Sequences

Suratna Budalakoti, University of California, Santa Cruz
Ashok N. Srivastava, Ph.D., NASA Ames Research Center
Ram Akella, Ph.D., University of California, Santa Cruz
Eugene Turkov, Research Institute of Advanced Computer Science

Abstract

This paper addresses the problem of detecting and describing anomalies in large sets of high-dimensional symbol sequences. [1] The approach taken uses unsupervised clustering of sequences using the normalized longest common subsequence (LCS) as a similarity measure, followed by detailed analysis of outliers to detect anomalies. As the LCS measure is expensive to compute, the first part of the paper discusses existing algorithms, such as the Hunt-Szymanski algorithm, that have low time-complexity. We then discuss why these algorithms often do not work well in practice and present a new hybrid algorithm for computing the LCS that, in our tests, outperforms the Hunt-Szymanski algorithm by a factor of five. The second part of the paper presents new algorithms for outlier analysis that provide comprehensible indicators as to why a particular sequence was deemed to be an outlier. The algorithm provide a coherent description to an analyst of the anomalies in the sequence, compared to more 'normal' sequences. The algorithms we present are general and domain-independent, so we discuss applications in related areas such as anomaly detection.

1 Introduction

We consider the problem of finding anomalies in a set of N discrete sequences $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, where each sequence $\mathbf{x}_i = \{x_{i1}, x_{i2}, ...x_{in_i}\}$ where the x_{ij} correspond to the jth symbol in the ith sequence. We assume all symbols are drawn from a finite but large alphabet $\mathcal{A}$ and that each sequence has a variable length n_i. We define an anomaly in the sequence to be a subsequence that differs significantly with respect to some measure from the majority of sequences in $\mathcal{X}$. We use the normalized longest common subsequence as the measure underlying an unsupervised clustering algorithm. Those sequences that are 'far away' from the majority of sequences in a cluster are called outliers or anomalies.

Although this is a general problem in anomaly detection in large sequences, we have built this methodology to address a key question in the aviation safety domain. We assume that we are given a set of sequences $\mathcal{X}$ that correspond to N landings of a specific aircraft model at a specific airport. The symbols that are recorded correspond to the switches in the cockpit of the airplane. As the pilot undergoes maneuvers to land the airplane, he or she flips switches and sets levers and other control mechanisms. The sequence of switches that a pilot flips during the course of the landing phase of the flight corresponds to the sequence x_i. Notice that in this scenario, since the duration of the landing phase can vary from flight to flight, and also since not all switches in the set $\mathcal{A}$ need to be flipped in each flight, the sequence length will be variable. We have tested our algorithms on sequences where the size of the alphabet is $|\mathcal{A}| \approx 1000$. A recent paper discusses the domain problem in more detail and elucidates some of the difficulties in addressing it using standard methods [11]. An example of the kind of anomalies the system targets for detection are mode awareness problems, such as confusion about the current state of cockpit automation.

Previous approaches to the task of anomaly detection focus on continuous sensor data, and do not distinguish discrete sensors from continuous. In this process they ignore the non-continuous as well as the sequential nature of the discrete sensors. In comparison, we focus on discrete sensors, specifically, sensors recording pilot actions, or switches. We are interested in the sequence in which the values for these sensors change during the course of a flight and finding anomalies in flight behavior based on this information.

[1] This work was supported by the NASA Aviation Safety Program, Aviation Systems Monitoring and Modeling element.

Our system performs two tasks, as part of the task of atypical events detection in flights: a) detection of atypical flights, b) finding events during the course of such flights that are anomalous or atypical. Task b) is important as each flight generates large amounts of data during its course, and simply identifying a flight as anomalous still leaves the problem of identifying the problem areas inside the flight unaddressed.

The first step of our approach is to merge data from all discrete sensors into a single sequence. This is done by recording a sensor only when it makes a transition. This transformation provides us with an event sequence for each flight. Following this, in the absence of training data, we treat the problem of finding atypical flights as an unsupervised learning problem. We first cluster the flights for each itinerary into groups, and identify the outliers in each cluster as atypical. We use the Longest Common Subsequence, a common measure in bioinformatics and Intrusion Detection systems, as the similarity measure for clustering flight data. We then present two new algorithms that use Bayesian Networks to efficiently identify anomalous events during the course of the flight.

We demonstrate the performance of these algorithms using operation information from about 10,000 flights, and developing the base clusters and locating anomalous flights by using these sequences.

A preliminary analysis of this problem may lead to a time-series based state-space approach, where at each time step, the current state of all symbols in the alphabet is recorded. For each sequence, we would obtain a matrix of size $(n_i \times |\mathcal{A})|$ in size. A decomposition of this matrix through an SVD approach could be used to generate a list of anomalies. The problem with this, and most other state-space approaches, is that the results are *independent* of the order of the data. Thus, the inherent pointers of phenomenon such as mode confusion can be lost.

Another approach that could be taken is to build Hidden Markov Models on the data sets and then analyze the likelihood of a particular sequence. While this approach maintains the sensitivity to the sequential nature of the data, it does not lend itself to interpretable models. Thus, we develop a new class of algorithms to detect anomalies and to present them in a comprehensible manner.

The area most closely related to the problem of anomaly detection in flight sequence data is that of anomaly detection. Anomaly detection in computer systems typically involves forming a model of normal behavior, and flagging any variation from the 'normal' model as an anomaly. The problem of anomaly detection in aircraft flight data is thus, by its nature related to the area of anomaly detection.

The approach we use to detect outliers is similar to some sequence analysis approaches for anomaly detection [7, 10]. Though superior results have been reported using the LCS metric in anomaly detection system [7], the same papers have also reported the cost of computing the LCS as a deterrent. Here,we present a faster algorithm for computing sequence similarity using LCS, which can help improve the speed of LCS based anomaly detection system. The amount of manual work required to be done by a system analyst is a large concern in the area of anomaly detection. The latter part of the paper present algorithms that accurately identify anomalous events inside a sequence. These algorithms can easily be adapted to the requirements of an anomaly detection system, and automate the task of identifying anomalies to a much larger extent, thereby cutting down the work required to be done by an analyst.

The area of sequence analysis and comparison owes much of its growth from its practical application in the area of bioinformatics. In this way, the new algorithms we present in this paper for sequence comparison and analysis are also related to the area of bioinformatics.

2 Outline of Approach

The main steps for the approach we use for anomaly detection in flight data is described below:

1. Cluster the sequences into groups, using the normalized common subsequence metric as the similarity measure. The clustering algorithm we use is the CLARA [1] k-medoids algorithm.

2. For each cluster, identify a certain percentage of the outliers as the anomalous sequences.

3. Do an analysis that tries to identify where and how the sequence deviates from normal behavior.

3 The Hybrid LCS Algorithm

We use the normalized longest common subsequence(nLCS) as the similarity measure for comparing flight sequences. Thus, given two sequences X and Y, of lengths m and n respectively, we calculate the normalized LCS by the formula:

$$nLCS = \frac{length(LCS)}{\sqrt{m \cdot n}}$$

Given two sequences X and Z, Z is a subsequence of X if removing some characters from X will produce Z. Z is a common subsequence of two sequences X and Y if Z is a subsequence of X and Y. The longest such subsequence between A and C shall be called the longest common subsequence (LCS). LCS is a very effective metric as it is able to detect similarity between two sequences without restricting itself to a location-based one-to-one match. The LCS metric has an optimal substructure property, which is the foundation of the well-known dynamic programming algorithm. Given two sequences X[m] and Y[n], the algorithm constructs a two-dimensional table L(m,n). An entry L(i,j) in the table gives the length of the LCS between the first i characters of X and the first j characters of Y. Figure 1 shows the table L constructed by the algorithm for two sample sequences. More information on the optimal substructure property and the dynamic programming algorithm can be found in [2].

The time complexity of the algorithm is m·n, which makes the algorithm very expensive over a certain sequence length. A vast amount of literature [3, 4, 5] exists that considers algorithms that attempt to improve the bounds on the computational time. A survey on LCS algorithms by Bergroth et al [6] divides such algorithms into three groups and compares their results empirically. The comparison turns up no clear winners. However, the group consisting of the algorithm introduced by Hunt and Szymanski [3] (and its variants), remain the most popular of the group, perhaps because they are relatively easy to implement.

3.1 The Hunt-Szymanski Algorithm

For completeness, we give the Hunt and Szymanski algorithm below. The algorithm we describe below gives just the length of the LCS. It does not give the actual LCS sequence, as we only need the length of the LCS to measure similarity. For the version of the algorithm that also gives the LCS, see [3] .

Algorithm 1: The Hunt-Szymanski Algorithm.

Input: Sequences X[m] and Y[n].
Output: Length of LCS.

Step 1: For each character in X, find where it occurs in Y. Store the information as a decreasing linked list array.
for i:= 1 to m
Set matchlist[i]:= $< j_1, j_2, \ldots, j_p >$*, such that* $j_1 > j_2 \ldots > j_p$ *and* $X[i] = Y[j_q]$*, for* $i \leq q \leq p$.

Step 2. Initialize the thresh array. Set all values to n+1.
for i:= 1 to n
thresh[i] := n+1.

Step 3. Compute thresh values through m iterations.
for i:= 1 to m
for each j on matchlist[i]
Find k s.t. $thresh[k-1] < j \leq thresh[k]$.
if $j < thresh[k]$
thresh[k]:=j.

Step 4. The lcs length is the largest k such that thresh[k] was updated.
$k :=$ *largest k such that* $thresh[k] \neq n+1$.
return k.

One way to understand the algorithm [3] is in term of successive computation of threshold values. Threshold $T_{i,k}$ = the smallest j such that X[1:i] and B[1:j] contain a common subsequence of length k. At the end of iteration i, the array thresh contains the values $T_{i,1\ldots n}$. Thus, at the end of m iterations, the LCS length is k such that T[k] has a valid value.

Another way to look at the algorithm is in terms of the two-dimensional table L created by the standard dynamic programming algorithm and how the Hunt-Szymanski algorithm relates to it. We first cover some standard definitions:

Given two sequences X and Y, (i, j) is said to be a *match* if $X[i] = Y[j]$.

A match (i,j) is called a *k-match*(or is said to have rank k) if the location $L(i, j) = k$.

A k-match (i,j) is called a *k-dominant-match* if, for all other pairs (i', j') of rank k, either $i' > i$ and $j' \leq j$, or $i' \leq i$ and $j' > j$. For example, in Figure 1, locations (3,2) and (4,4) are dominant matches.

In light of these definitions, we can say that, at the end of i iterations, the array thresh contains the location of the k-dominant-match at location thresh[k].

The Hunt-Szymanski(HS) algorithm for two sequences X[m] and Y[n] can now be understood as follows:
Algorithm 2: The Hunt-Szymanski Algorithm in terms of dominant matches.

For $i = 1$ *to m,*
For each match (x_i, y_j),y_j *in decreasing order,*
if y_j *lies between the* k^{th}
and $(k+1)^{st}$ *dominant match,*
move the $(k+1)^{st}$ *dominant match location to* y_j.

X \ Y	1 A	2 G	3 T	4 G	5 G	6 C	7 T	8 C	9 G	10 T	11 T	12 A
G 1	0	1	1	1	1	1	1	1	1	1	1	1
A 2	1	1	1	1	1	1	1	1	1	1	1	2
G 3	1	2	2	2	2	2	2	2	2	2	2	2
G 4	1	2	2	3	3	3	3	3	3	3	3	3
T 5	1	2	3	3	3	3	4	4	4	4	4	4
G 6	1	2	3	4	4	4	4	4	5	5	5	5
C 7	1	2	3	4	4	5	5	5	5	5	5	5
A 8	1	2	3	4	4	5	5	5	5	5	5	6

Figure 1: Example: Longest common subsequence table.

For example, suppose we are processing X[3] = 'G' in example figure 1, using the HS algorithm. Looking at the indices backwards, we find the first match for G at location 9. We see that this match lies between the current dominant 1-match(at 1) and 2-match(at 12). This is done by searching through 'thresh', in step 3, where it will find that thresh[1]=1 and thresh[2]=12. We then move forward the 2-dominant match to 9. For the table L, this can be done by setting all values between 9 and 12 to 2. The HS algorithm does this by setting thresh[2] from 12 to 9. The reason why the algorithm tries to move dominant matches forward is to create space for more dominant matches. The later a dominant match occurs in sequence Y, the less space is there for more matches of X with Y.

The computation complexity of the HS algorithm can now be calculated as follows. Assuming both sequences are of equal length(n), it searches through array thresh of length n, r times, where r is the total number of matches between the two sequences. Ignoring the time required to create the index, the complexity of the HS algorithm is given by $O(r \cdot \log n)$.

If σ is the set of unique characters(symbols) over both sequences X and Y, and if there are k such unique characters, the value of r is given by

$$r = \sum_{i=1}^{k} \eta(\sigma_i^X) \cdot \eta(\sigma_i^Y)$$

where, $\eta(\sigma_i^X)$ represents the count of character σ_i in sequence X.

As is clear from the expression, the value of r is small when the σ frequencies are small, or when the frequency distributions for σ^X and σ^Y are different. However, if the sequences being compared are drawn from the same dataset and represent the same phenomenon, this is unlikely to be the case.

In fact, in most areas where the LCS metric is used, these conditions seldom hold. For example, in bioinformatics, the frequency distribution of characters is uniform and the character size is small(4-8). In anomaly detection [7, 8], system calls and user commands frequencies can be modeled as Zipf-like distributions. In Web Usage Mining [9], another area where LCS can be used as a measure, web traffic is known to show a Zipf-like frequency distribution. Again, in text matching, language characters and words are known to follow a Zipf-like distribution. Clearly the HS algorithm does not perform optimally for Zipf-like distributions, where certain $\eta(\sigma_i)$s are very large compared to others.

A number of methods have been devised to reduce the actual value of r in the term $O(r \cdot \log n)$. A simple one, which works very well in practice, will be to search through 'thresh' only if we crossed a dominant match between two successive index locations. This can be done simply by first checking if the location k updated in the previous iteration satisfies $thresh[k-1] < j' \leq thresh[k]$, where j' is the current index location being processed. For characters with a large value of r, this cuts down the number of searches by quite a margin. However, this still means we have to search through the entire thresh array for each search. This search is more difficult to optimize.

3.2 The Hybrid Algorithm

One of the reasons that the HS algorithm does not perform better in many cases is because its data structures provide it with very little information. We next present a new algorithm that attempts to make up for this drawback of the HS algorithm. The algorithm may be considered a hybrid of the HS algorithm and the standard dynamic programming algorithm[2]. This is because, it essentially follows the procedure described in Algorithm 2 above. However, it maintains a different data structure, an array 'row' which, at any iteration i of the algorithm, corresponds to the row L(i, 1:n) of the standard dynamic programming algorithm.

We give the hybrid LCS algorithm below:

Algorithm 3: Hybrid Algorithm for LCS.

Input: Sequences X[m] and Y[n].
Output: Length of LCS.

Step 1: For each character in X, find where it occurs in Y. Store the information as a decreasing linked list array.
for i:= 1 to m
Set matchlist[i]:= $< j_1, j_2, \ldots, j_p >$*, such that* $j_1 > j_2 \ldots > j_p$*, and* $X[i] = Y[j_q]$ *for* $i \leq q \leq p$.

Step 2. Initialize the row array. Set all values to 0.
for i:= 1 to n, row[i] := 0.

Step 3. Compute L(1,1:n) at each iteration i.
Set prev_index := n.
for i:= 1 to m
for each j on matchlist[i]
Update the k-match value of j.
Set row[j] := row[j-1] + 1.
Step 3.a. Check if the dominant match is located right next to j. In that case, array to search length = 0, so skip.
if row[j] = row[j+1]
prev_index = j.
skip.
Step 3.b. Else find the match and move it to j.
find $j < k \leq$ *prev_index s.t.*
row[k] = row[j].
if $\exists k$
row[j+1 ... k] = row[j].
prev_index = j.

Step 4. The last value in row gives the lcs length.
k := row[n].
return k.

Let us now see how the hybrid algorithm processes X[3] = 'G' in example figure 1. At the start of iteration 3, 'row' will look exactly like L(2,:) of the example. The algorithm will find the match j at location 9. It will increment row[9] to 2. Since $row[9] > row[10]$ now, it will search through (9,12) for the 2-dominant-match. It will find the value 2 at location 12, which tells it that a 2-dominant-match is placed at location 12 in one of the previous rows. It will now move the 2-dominant match to 9, by setting $row[10 \ldots 12]$ to 2. It will then set prev_index to 9, and move to the next location, and so on.

The complexity of the algorithm can be calculated as follows: We ignore the time taken to create index, as with the HS algorithm. Step 2 takes linear time. Step 3.a is executed r times. Assuming a uniform distribution of σ, we can assume the matches r are distributed uniformly across row. In that case step 3.b is executed n $\cdot$ (min(r / n,count(dominant matches))). As the maximum count of dominant matches is equal to the LCS, we can say that Step 3b is executed $n \cdot (min(r/n, LCS))$ times. Each time we do a binary search through a small portion of the sorted array row. Assuming that the matches are evenly distributed across each row, we say that the average length of array row is $\log(n^2/r)$:

$$O(n \cdot min(\frac{r}{n}, LCS) \cdot \log(\frac{n^2}{r}))$$

It is obvious that this value shall generally be much smaller than $r \cdot \log n$, and can never be larger.

However, the time complexity computed is only in terms of number of comparisons. The algorithm performs more read-writes than the HS algorithm. In the worst case, the time complexity of the number of read-writes performed is $O(n^2)$. However, we find that the cost of memory writes is negligible in practice. Also, as most of the writes consist of the same value written over contiguous areas inside an array, a clever programmer can easily find ways to optimize these writes. But our current implementation makes no such optimizations.

Table 1 and 2 compare the computation running time of the LCS algorithms described above. The test data we use is generated exactly the methodology followed in the survey paper on LCS algorithms by Bergroth et al [6]. More information on the test data generation methodology can be found in Section 5. Our results clearly show that the hybrid algorithm runs many orders faster than the HS algorithm.

4 Outlier Analysis

For data embedded in a vector space, or following a known statistical distribution, once the outliers have been identified, the reasons why these particular data points were considered outliers is generally easy to answer. For example, in the vector space model, if a data point is an outlier, it is because it has an abnormally high/low value along one or more dimensions. Such properties are easy to detect. However, in the case of sequential data, even after the identification of outliers, it is often difficult to say, by simple obser-

	Sequence Lengths(characters)						
Algorithm Name	500	1000	2000	4000	10,000	20,000	40,000
Hybrid Algorithm	0.001	0.004	0.02	0.06	0.50	3.30	14.00
Hunt-Szymanski Algorithm	0.006	0.02	0.06	0.25	1.60	8.00	33.40
Std. Dynamic Programming	0.007	0.03	0.14	0.60	3.90	17.00	114.50

Table 1: Running time of LCS algorithms(seconds)- σ distribution is Zipf

	Sequence Lengths(characters)						
Algorithm Name	500	1000	2000	4000	10,000	20,000	40,000
Hybrid Algorithm	0.002	0.008	0.036	0.14	1.21	4.90	27.20
Hunt-Szymanski Algorithm	0.01	0.038	0.166	0.52	4.85	20.90	96.00
Std. Dynamic Programming	0.009	0.032	0.14	0.64	5.10	18.10	109.20

Table 2: Running time of LCS algorithms(seconds)- σ distribution is uniform

vation or analysis, why the data point was considered an outlier.

But simply declaring a sequence as anomalous is not very useful for an analyst. Manually analyzing a sequence to discover why it was considered anomalous is a taxing and time-consuming activity, particularly if the sequences are long and drawn from a large character size. This section describes algorithms that attempt to automate this activity as far as possible. Our system, once it classifies a sequence as anomalous, analyzes this sequence further, and provides detailed information to the analyst about the atypical events inside the flight sequence. This information can be classified into (1) A sequence of switches was expected at a given stage but were not flipped. (2) A sequence of switches was not expected at a given stage but were flipped. (3) A sequence of switches were flipped in the wrong order.

The approach we take to identify such anomalies inside a sequence consists of the following steps:

Define/derive an objective function to maximize over the outlier sequence. We define an objective function F for the outlier sequence, maximizing which, we expect, will maximize the likelihood that the sequence is not an outlier member of its cluster. A reasonable objective function is the average, or the weighted mean, of the similarity score(nLCS) of the outlier sequence with all the sequences in the cluster. Alternatively, we construct a generative model for the sequences in a cluster, and derive F from this model.

Identify possible changes to the sequence. We next try to identify the changes(insertions/deletions) that, if made to the outlier sequence, will maximize this objective function F.

We now divide anomalies to be identified inside a sequence into two categories, and define them in terms of the objective function O.

Non-essential Character: A character at a location in a sequence is said to be non-essential to the sequence if its removal from that location will improve the objective function score F for the sequence, in relation to its cluster.

Missing Character: A character is said to be missing from a particular location in a sequence if its addition at the location will improve the objective function score F for the sequence, in relation to its cluster.

The problem of finding the anomalous areas in a sequence can thus be divided into two parts: a) find all the non-essential characters inside the anomalous sequence, and b) find all the missing characters from the sequence.

The next section discusses the objective function F.

4.1 The Objective Function

4.1.1 Bayesian Network based model of F

A generative model for a cluster can be constructed in terms of a simple Bayesian Network. As part of this network, each of the sequences in the cluster is generated by the centroid(identified during the clustering stage) by a certain probability. The probability of generation of a sequence from the centroid can be taken as proportional to the value of the normalized LCS score between the sequence and the centroid. The outlier sequence can be considered as being gen-

erated by each of the sequences of the cluster by a certain probability. We assume that the probability of generation of the outlier from each sequence is proportional to the normalized LCS score between the sequence and the outlier.

We now calculate the changes required to be made to the outlier sequence, that maximize the probability that the outlier sequence was generated from the cluster.

Let C be the centroid, the sequences inside the cluster be represented by $S_1, S_2, \ldots, s_n$, and O be the outlier. In that case, we want to maximize $P(C/O)$.

$$P(C|O) \propto P(O \wedge C)$$

$$P(O \wedge C) = P(C) \sum_{i=1}^{N} P(O|S_i) \cdot P(S_i|C)$$

$$\propto \sum_{i=1}^{N} P(O|S_i) \cdot P(S_i|C)$$

Since $P(O|S_i) \propto nLCS(O, S_i)$ and $P(S_i|C) \propto nLCS(C, S_i)$, we get

$$P(C|O) \propto \sum_{i=1}^{N} nLCS(O, S_i) \cdot nLCS(C, S_i)$$

Hence, in the case of a Bayesian Network based model for a cluster, the objective function to be maximized is given by

$$F'(O, C) = \sum_{i=1}^{N} nLCS(O, S_i) \cdot nLCS(C, S_i)$$

$$= \sum_{i=1}^{N} \frac{LCS(O, S_i)}{\sqrt{l_O \cdot l_{S_i}}} \cdot \frac{LCS(S_i, C)}{\sqrt{l_{S_i} \cdot l_C}}$$

$$= \frac{1}{\sqrt{l_O \cdot l_C}} \cdot \sum_{i=1}^{N} \frac{lcs(O, S_i) \cdot lcs(S_i, C)}{l_{S_i}}$$

$$= \frac{\beta}{\sqrt{l_O}} \cdot \sum_{i=1}^{N} \frac{lcs(O, S_i) \cdot lcs(S_i, C)}{l_{S_i}}$$

Ignoring β, the objective function is given by

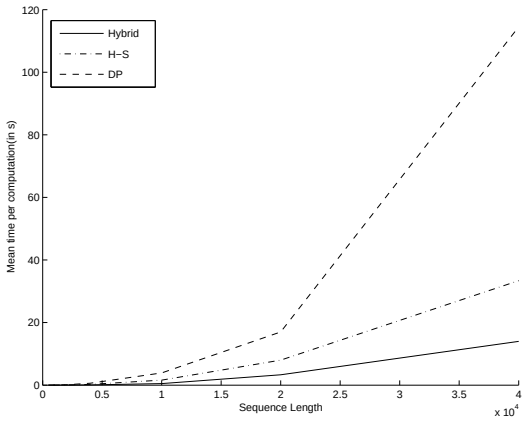

Figure 2: Time to compute LCS - Zipf Distribution

$$F(O, C) = \frac{1}{\sqrt{l_O}} \cdot \sum_{i=1}^{N} \frac{lcs(O, S_i) \cdot lcs(S_i, C)}{l_{S_i}}$$

In general, a Bayesian Network based model based objective function is more effective when the cluster is large can be said to contains small sub-clusters, as the Bayes net model optimizes with respect to the sequences most similar to the outlier.

4.1.2 Mean / Weighted Mean based Objective Function

Given an outlier sequence O and a cluster C, a weighted mean based objective function F can be given by:

$$F(O, C) = \sum_{i=1}^{N} \lambda_i \cdot nLCS(O, S_i)$$

Here N is the number of sequences in the cluster. The weight λ can be set as equal for all sequences, or alternatively, it may be set as proportional to the score the sequence with the centroid.

Expanding the value of $nLCS(O, S_i)$, we get

$$F(O, C) = \frac{1}{\sqrt{l_O}} \cdot \sum_{i=1}^{N} \lambda_i \cdot \frac{LCS(O, S_i)}{\sqrt{l_{S_i}}}$$

It should be better to use the weighted mean based objective function when the clusters are small and homogenous, as it assumes that the outlier should resemble all sequences in the cluster equally closely.

4.2 Maximizing the objective function

We discuss the algorithms used to find the changes that would maximize the objective function, with the condition that *each individual change by itself increases the value of objective function.* The constraint is important as an algorithm that does not follow this constraint might go far beyond detecting anomalies in the sequence, and start suggesting changes which change the very nature and character of the sequence. For example, it might remove every character from the outlier sequence and replace it by characters from the cluster centroid. This might maximize the objective function, but it shall not provide any useful information on why the sequence was considered anomalous. Hence, it is important that we add this constraint to our analysis. The algorithms we discuss below are based on the objective function derived for the Bayesian Network based model of a cluster. The same algorithms can be applied, with minor modifications, to the case where the objective function is defined as a weighted mean.

Suppose we removed a character c from a location l inside outlier sequence O. The impact would be as follows:

1. The length of the outlier sequence would decrease by 1.
2. For the sequences S_i where c is part of their LCS with O, the LCS value will decrease by 1.
3. The LCS value will remain unchanged for all other sequences in the cluster.

The objective function is given by:

$$F(O,C) = \frac{1}{\sqrt{l_O}} \cdot \sum_{i=1}^{N} \frac{lcs(O,S_i) \cdot lcs(S_i,C)}{l_{S_i}}$$

Let the new F(O,C) as a result of this change be represented by F'(O,C). F'(O,C) is given by:

$$F'(O,C) = \frac{1}{\sqrt{l_O - 1}} \{ \sum_{c \in lcs(O,S_j)} \frac{(lcs(O,S_j) - 1) \cdot lcs(S_j,C)}{l_{S_j}}$$

$$+ \sum_{c \notin lcs(O,S_k)} \frac{lcs(O,S_k) \cdot lcs(S_k,C)}{l_{S_k}} \}$$

$$= \frac{1}{\sqrt{l_O - 1}} \cdot \{ \sum_{i=1}^{N} \frac{lcs(O,S_i) \cdot lcs(S_i,C)}{l_{S_i}}$$

$$- \sum_{c \in lcs(O,S_j)} \frac{lcs(S_j,C)}{l_{S_j}} \}$$

$$\text{Let} \sum_{c \in lcs(O,S_j)} \frac{lcs(S_j,C)}{l_{S_j}} = b_c$$

Replacing b_c and F(O,C) in the equation for F'(O,C) gives:

$$F'(O,C) = \frac{1}{\sqrt{l_O - 1}} \cdot (\sqrt{l_O} \cdot F(O,C) - b_c)$$

It can be shown that, given k characters at different locations, all with same value of $b_c = b$:

$$F'(O,C) = \frac{1}{\sqrt{l_O - k}} \cdot (\sqrt{l_O} \cdot F(O,C) - k \cdot b)$$

4.2.1 Detecting non-essential characters

We can now present a simple greedy algorithm to find all characters removing which shall improve the objective function score. The algorithms we describe follow a greedy strategy. The constraint that each change to a sequence should by itself improve the score F, makes the problem most suitable for solution using a greedy algorithm. We discuss the optimality of these algorithms in Section 4.3.

Algorithm 4: Non-essential character Detection Algorithm.

Input: Outlier sequence O and cluster C with sequences $< S_1, \ldots, S_n >$
Output: N ,the list of non-essential characters in O.
Step 1:Declare array b[l], where l is the length of O.
for i:= 1 to l, b[i] := 0.

Step 2:Calculate F, and b_c *for each character c in O.*
for i:= 1 to n
Get the LCS of O with S_i*. For each* $c \in LCS(O,S_i)$,
Set $b[c] := b[c] + lcs(O,S_i)/l_{S_i}$.
Set $F = F + lcs(O,S_i) \cdot lcs(C,S_i)$.

Step 3: Find the next character to be replaced.
Find $b = min\ (b_c)$.
Find all C, such that, $b_c = b$.
Set k := Size(C).

Step 4: Calculate new value of F.

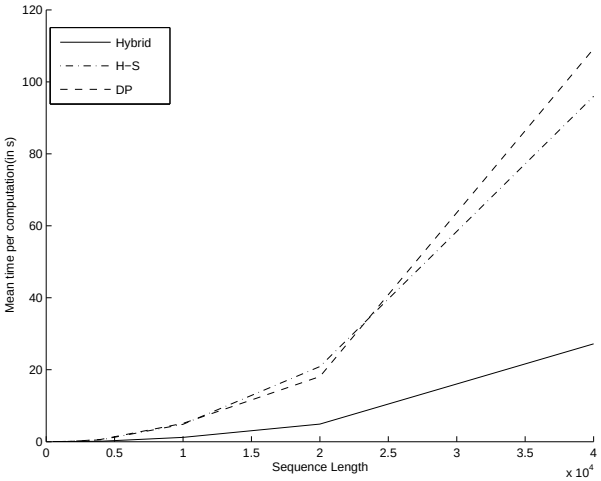

Figure 3: Time to compute LCS - Uniform Distribution

Set F_old = F.
$F = \frac{1}{\sqrt{l_O - k}} \cdot (\sqrt{l_O} \cdot F - k \cdot b)$
If F > F_old
Add c ∈ C to N.
Set b[c] = max+1, for all c ∈ C.
Go to Step 3.

Step 5. All non-essential characters are stored in N.
Return N.

4.2.2 Detecting missing characters

This section focuses on the problem of finding the missing characters from a sequence. An analysis similar to the one above shows that adding a character σ to a location c in an outlier sequence O sequence changes the objective function F as follows:

$$F'(O,C) = \frac{1}{\sqrt{l_O + 1}} \cdot (\sqrt{l_O} \cdot F(O,C) + b_{c\sigma})$$

Here the value of $b_{c\sigma}$ is given by the following expression:

$$b_{c\sigma} = \sum_{i=1}^{m} \frac{lcs(S_i, C)}{l_{S_i}}$$

Here m is the number of sequences in the cluster will contain the symbol σ as part of their LCS with O, if it is added between location c-1 and c of sequence O.

Similarly, adding k symbols to a sequence at various locations, such that for all k symbols, $b_{c\sigma}$ = b, will give change F as follows:

$$F'(O,C) = \frac{1}{\sqrt{l_O + k}} \cdot (\sqrt{l_O} \cdot F(O,C) + k \cdot b$$

We provide below a greedy algorithm based on the above formulae:

Algorithm 5: Missing characters Detection Algorithm.

Input: Outlier sequence O and cluster C with sequences $< S_1, \ldots, S_n >$
$N_{symbols}$*: the total number of unique symbols in the sequence.*
Output: M ,the list of missing characters in O.

Step 1:Declare array b[l], where l is the length of O.
for i:= 1 to l
for j:=1 to $N_{symbols}$*, b[i][j] := 0*

Step 2:Calculate F, and b_{cp} *for O.*
for i:= 1 to n
Get the LCS of O with S_i*.*
Foreach $c_j \in LCS(O, S_i)$*,*
Set $b[c][k] := b[c][k] + lcs(O, S_i)/l_{S_i}$*.*
where k occurs in S_i *between* c_{j-1} *and* c_j*.*
Set $F = F + lcs(O, S_i) \cdot lcs(C, S_i)$*.*

Step 3: Find the next set of missing characters.
Find b = max (b).
Find C = all (i,j) pairs, such that, b(i,j) = b.
Set k := Number of such (i,j) pairs.

Step 4: Calculate new value of F.
Set F_old = F.
$F = \frac{1}{\sqrt{l_O + k}} \cdot (\sqrt{l_O} \cdot F + k \cdot b)$
If F > F_old
Add (i,j) ∈ C to M.
Set b(i,j)=0 for all $(i,j) \in C$*.*
Go to Step 3.

Step 5. All missing characters and their locations are stored in M.
Return M.

4.2.3 Reconstructing missing character sequences

The algorithm given above can only detect the symbols that should be inserted between two characters in a sequence, but cannot detect the order in which they should be inserted. For example, the algorithm might say that characters D and E should be inserted between characters A and C in a given outlier sequence, but not whether DE or ED should be inserted.

There are two possible approaches towards an algorithm which shall detect the exact sequence in which characters should be inserted. The naive approach is to actually insert the character in the outlier sequence each time the addition algorithm prescribes an addition, and recalculating the array b accordingly. Updating b might involve re-comparing the modified outlier sequence with all the sequences in the cluster.

An alternative approach, which minimizes the number of comparisons between the outlier and the cluster, is to allow extra characters to creep into the sequence, and, at the end, make a call to Algorithm 3(section 4.2.1), which detects and removes these extra characters.

The algorithm is based on the observation that, for a given sequence of length l, adding a symbol *sigma* before a character c will increase the score F of the sequence with respect to the cluster only if the value of $b_{c\sigma}$ is above the certain threshold. This threshold can be derived to be:

$$b_{c\sigma} > T = (\sqrt{l+1} - \sqrt{l}) \cdot F$$

If every insertion we make to the sequence is a valid insertion, that is , it is greater than the threshold described above, the value of this threshold increases with every insertion. Thus, at a given sequence length l, if we pick all (c, σ) pairs such that $b_{c\sigma}$ is greater than the threshold at l, we have picked all the possible candidates for insertion in the future, as the threshold value can only increase. Due to lack of space, we only give a short outline of the algorithm.

Algorithm 6: Algorithm to compute missing character sequences.

Step 1: Given an outlier sequence O of length l and cluster C, calculate the array $b(l, \eta(\sigma))$ and the current value of F, as in Step 2 of algorithm 5.

Step 2: Calculate the current value of the threshold T.

Step 3: Set $O' := O$. Set $b' = b$. Find all $Candidates = b(i,j) > T$.

Step 4: Before each row i of b, find character σ_j, if it exists, such that $b(i,j) \in Candidates$ and $b(i,j) = max(b(i^{th}row)$. Insert σ_j in O', at the location in O' for which b' has the maximum value for σ_j, between the rows corresponding to i-1 and i in b. Set $b(i,j) = 0$.

Step 5: Calculate $b'(l, \eta(\sigma))$ for the new O, , as in Step 2 of algorithm 5. Go to Step 4.

Step 6: Call Algorithm 4 with O and C, to remove inessential characters that were added. Compare O_{old} and O to get the list of additions.

Essentially, Step 4 and 5 combine to find what the final sequence would look like if all the candidates were inserted into the sequence. These steps work as follows:

Suppose we have an outlier sequence ABCD, and step 3 finds characters E,F and G, above T between A and B, such that b(B,E) > b(B,F) > b(B,G), and $I > T$ between C and D(here b'(B,E) is read as the score in array b' for character E, for its insertion before location B in ABCD). In the first iteration, the algorithm will insert E between A and B, and I between C and D. It will then recalculate the $b'_{c\sigma}$ array for AEBCID. It will then check if F has a higher score before E or after E, and insert F accordingly. Suppose the new sequence is AFEBCID. It will then check whether G has a higher value in b' between AF, FE, or EB and insert G there. The final sequence may look like AFGEBCID.

Step 6 calls Algorithm 4 to remove the extra characters from O'. Supposing it is found that F is a non-essential character in the sequence and removed. However, G is still in the right location because, if F had never been inserted, b'(E,G) would still be greater than b'(B,G),as the same relation held when F was inserted between A and E, even though the insertion of F reduced the value of b'(E,G).

4.3 Discussion: Optimality

We say an algorithm for detecting non-essential/missing characters is optimal if it gives the exact set of deletions/insertions which will maximize the value of F, with the constraint that each modification to the sequence by itself increases the value of F.

We describe below an optimal strategy for identi-

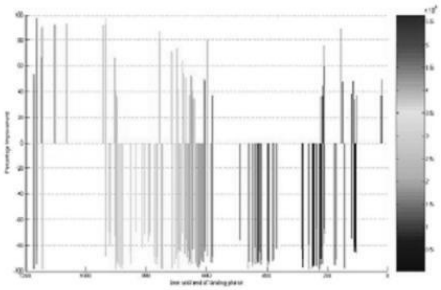

Figure 4: Sample: Generated graph for an atypical flight.

fying optimal deletions, for a given outlier sequence O of length l and a cluster S of sequences. A similar strategy can be constructed for insertions. a) Calculate the b_c array. b) Remove the character from O with b, the maximum value of b_c. c) If the previous step does not increase the value of the objective function, exit. d) Recalculate the b_c array for the new sequence O, that does not contain the removed character.

This is an optimal strategy, because the increase in value of F is directly proportional to the value of b[c] for the removed character c. Suppose we had to remove only one character from the sequence. It is obvious that removing the character with the maximum value of b[c] would be optimal. This reduces the problem to one of finding the best character to remove from the new sequence O, with length l-1. Clearly, the same algorithm can be applied again. We must stop if, in step c), the value of F does not increase, due to the constraint discussed earlier. Thus we can see that a greedy algorithm as described above will give the optimal set of deletions. However, computing b_c after every insertion/deletion is expensive, and the algorithms described above compute b_c only once. The approach shall be optimal if the array b_c does not change considerably between iterations. However, this may not always be true. An alternative is to re-calculate b_c once again after the algorithm ends, and rerun the algorithm, to identify any deletions which were erroneously missed in the first pass, and the insertion algorithm(algorithm 5) to find erroneously deleted characters.

5 Experimental Results

The two datasets used to compare the LCS algorithms was generated synthetically, using exactly the procedure used by Bergroth et al[6] in their survey, to facilitate comparison with existing LCS algorithms. The first dataset consisted of sequences drawn from a uniform distribution with $\sigma = 8$. The second dataset was drawn from a Zipf distribution with $\sigma = 256$. While [6] tested only with sequences of length 4000, we tested over a much larger range of sequence lengths, between 500 and 40,000. The results of the comparison are presented in Tables 1 and 2. Figures 2 and 3 provide plots of the same data. As the results clearly show, the new hybrid algorithm outperforms the other two algorithm by many times.

The clustering and outlier analysis algorithms were used over a dataset consisting of the landing phase sequence information for 6400 flights. The sequence dataset consisted of 6400 distinct sequences, varying widely in length from 800 to over 9000. The number of distinct symbols(σ) was around 1000. The average sequence length was approximately 1500. The number of distinct clusters in the data was empirically estimated to be three, which was the value passed to the k-medoids algorithm. The clustering is currently implemented in MATLAB and takes around 2 hours. It is not expected to take more than 10 minutes in a C implementation.

Our outlier analysis algorithms output the following information: *A graphical representation of the anomalous areas inside an outlier sequence.* A sample graph for an atypical flight is presented in Figure 4. The horizontal axis, represents the time remaining until the plane's tires touched the runway. The positive direction of the vertical axis represents the suggested insertions, that is, switches the algorithm thinks the pilots should have pressed at that stage. The negative direction represents the suggested deletions, that is, the switches the algorithm believes the pilots should not have pressed at that stage, but did. The height of the columns indicates the confidence the algorithm has in its prediction, which is measured as proportional to the improvement in the score of the objective function F. A graph with no bars would be representative of a completely normal flight. The graph provides a simple visual interface that allows the analyst to focus his interest on the areas which are suspected to be most anomalous, and saving him the trouble of analyzing the entire sequence. *A detailed report describing these anomalous areas.* A report is generated in parallel by the system, which gives detailed information on the switches that the algorithm

believes should have been pressed/not pressed at that stage.

6 Conclusions

The paper describes a system designed with the aim of detecting anomalies in discrete flight data. It does so by clustering flight data sequences using the normalized longest common subsequence(nLCS) as the similarity measure. As the nLCS is expensive to compute, we discussed algorithms, such as the Hunt-Szymanski algorithm, which have lesser runtime complexity. We also discussed why they do not work so well in actual practice, and presented a new hybrid algorithm,which is many orders faster than the Hunt-Szymanski algorithm. This is an important contribution as the LCS is a commonly used algorithm in many areas and the running time of LCS algorithms is a frequent bottleneck. Following this, we presented algorithms, based on a Bayesian model of a sequence cluster, that detect anomalies inside sequences. In doing this, we move beyond what most current anomaly detection systems achieve, in not only predicting which sequences are anomalous, but by providing explanations as to why these particular sequences are anomalous. Our approach is general and not restricted in any way to a domain, and these algorithms can be of interest in other areas such as anomaly detection and event mining.

References

[1] L. Kaufman and P.J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*, John Wiley and Sons, Inc., New York (1990)

[2] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to algorithms*, The MIT Press; 2nd edition.

[3] James W. Hunt and Thomas G. Szymanski, *A Fast Algorithm for computing Longest Common Subsequences*, Communications of the ACM, Volume 20, Issue 5 (May 1977),Pages: 350 - 353.

[4] D. S. Hirschberg, *Algorithms for the Longest Common Subsequence Problem*, Journal of the ACM, Volume 24, Issue 4 (October 1977),Pages: 664 - 675.

[5] D. S. Hirschberg, *A Linear Space Algorithm for computing Maximal Common Subsequences*, Communications of the ACM, Volume 18, Issue 6 (June 1975),Pages: 341 - 343.

[6] L. Bergroth, H. Hakonen and T. Raita, *A Survey of Longest Common Subsequence Algorithms*, Proceedings of the Seventh International Symposium on String Processing Information Retrieval(SPIRE), 2000.

[7] K. Sequeira and M. Zaki, *ADMIT: Anomaly based Data Mining for Intrusions*, Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(SIGKDD), 2002.

[8] Scott Coull, Joel Branch and Boleslaw Szymanski, *Intrusion Detection: A Bioinformatics Approach*, Proceedings of the 19th Annual Computer Security Applications Conference(ACSAC), 2003.

[9] A. Banerjee and J. Ghosh, *Clickstream Clustering using Weighted Longest Common Subsequence*, Proceedings of the 1st SIAM International Conference on Data Mining (SDM): Workshop on WebMining, 2001.

[10] T. Lane and C. Brodley, *Temporal sequence learning and data reduction for anomaly detection*, ACM Transactions on Information and System Security (TISSEC), Volume 2, Issue 3 (August 1999), Pages: 295 - 331.

[11] A. N. Srivastava, *Discovering System Health Anomalies using Data Mining Techniques*, Proceedings of the 2005 Joint Army Navy NASA Airforce Conference on Propulsion, 2005.